Dolly: Virtualization-driven Database Provisioning for the Cloud

Emmanuel Cecchet, Rahul Singh, Upendra Sharma, Prashant Shenoy

University of Massachusetts, Amherst, USA

{cecchet,rahul,upendra,shenoy}@cs.umass.edu

Abstract

Cloud computing platforms are becoming increasingly popular for e-commerce applications that can be scaled on-demand in a very cost effective way. Dynamic provisioning is used to autonomously add capacity in multi-tier cloud-based applications that see workload increases. While many solutions exist to provision tiers with little or no state in applications, the database tier remains problematic for dynamic provisioning due to the need to replicate its large disk state.

In this paper, we explore virtual machine (VM) cloning techniques to spawn database replicas and address the challenges of provisioning shared-nothing replicated databases in the cloud. We argue that being able to determine state replication time is crucial for provisioning databases and show that VM cloning provides this property. We propose Dolly, a database provisioning system based on VM cloning and cost models to adapt the provisioning policy to the cloud infrastructure specifics and application requirements. We present an implementation of Dolly in a commercial-grade replication middleware and evaluate database provisioning strategies for a TPC-W workload on a private cloud and on Amazon EC2. By being aware of VM-based state replication cost, Dolly can solve the challenge of automated provisioning for replicated databases on cloud platforms.

Categories and Subject Descriptors D.2.9 [**Software Engineering**]: Management.

General Terms Algorithms, Management, Measurement, Performance, Design, Experimentation.

Keywords Database, Autonomic Provisioning, Virtualization.

1. Introduction

Online applications have become popular in a variety of domains such as e-retail, banking, finance, news, and social networking. Typically such web-based applications are hosted in data-centers or on cloud computing platforms, which provide storage and computing resources to these applications. Numerous studies have shown that the workloads seen by these web-based cloud applications are highly dynamic and exhibit variations at different time-scales [21], [22]. For instance, an application may see a rapid increase in its popularity, causing its workload to grow sharply over a period of days or weeks. At shorter time-scales, a flash crowd can cause the application workload to surge within minutes. Applications can also see seasonal trends such as higher workloads during particular periods, e.g., during Black Friday, marketing campaigns, or a new product launch.

One possible approach for handling workload fluctuations is to employ dynamic provisioning of server capacity. Dynamic provisioning involves increasing or decreasing the number of servers (and server capacity) allocated to an application in response to workload changes. Dynamic provisioning is especially well-suited to web-based cloud applications for two reasons. First, it is often difficult to estimate the peak workload of an Internet application, making it challenging to a priori provision for the peak demand. Second, today's cloud platforms support on-demand allocation of servers and employ a pay-as-you-go service model. These features are attractive from an application provider's perspective, since servers can be requested only when a workload spike arrives or is anticipated, and charging is based only on the duration of the workload surge. Cloud platforms employ virtualization to support these features—upon a customer request for a new (virtual) server, a new virtual machine (VM) is created on a physical server with idle capacity, and the specified virtual disk image is copied to the server, upon which the server is ready for use. In fact, cloud platforms such as Amazon's EC2 platform already support dynamic provisioning (aka "auto scaling") where such VMs are automatically started when a threshold on a user-specified metric such as CPU utilization is exceeded in the current application [1].

Much of the prior work on dynamic provisioning [20], [21], [22], [4] has assumed that web applications have a multi-tier architecture and focus on dynamic provisioning of the front web tier or the middle application tier. Provisioning of these front and middle tiers is simple since these tiers have little or no application state and provisioning merely involves dynamic startup (or shutdown) of VMs in response to workload fluctuations. This prior work assumes that the backend database tier, where much of the application state is stored, is over-provisioned and thus does not require dynamic provisioning. However, in scenarios where the database tier is the bottleneck (e.g., due to compute-intensive query workloads), this simplifying assumption has meant that our inability to a priori estimate the peak workload for Internet applications will cause the database tier to become overloaded and drop user requests. Further it prevents the web application from fully exploiting the benefits of the pay-as-you-go and on-demand

VEE'11, March 9–11, 2011, Newport Beach, California, USA.

server allocation in the cloud for the backend tier. Dynamic provisioning of the back-end database tier has not been considered in the prior literature since it is harder to implement—replication and synchronization of the associated disk state of the database needs to be handled, in addition to the 'simpler' problem of starting up new database VM replicas.

In this paper, we consider the problem of dynamic provisioning of the database tier of online web applications. We use virtualization as a key building block of our dynamic provisioning system, in particular by leveraging VM snapshots and cloning as the basis for replicating database state in a platform-independent manner. In addition, we devise intelligent state replication strategies to reduce the latency of starting up new database replicas in virtualized public and private clouds.

1.1 Why is database provisioning hard?

Dynamic provisioning of server capacity typically involves two problems: *when* to trigger a capacity increase (or decrease), and *how* to achieve the desired capacity addition or reduction. Both the "when" and the "how" questions are simpler in case of the web and application tiers than the database tier.

Typically the decision of *when* to trigger provisioning is made in a *lazy* fashion for the web and the application tiers—upon an actual significant workload change, or an anticipation of one in the near future. Such lazy triggers are appropriate for these tiers since front-end provisioning schemes assume that new capacity can be added immediately whenever needed and that the only latency is that incurred for VM startup. In contrast, provisioning of a new database replica involves (i) extracting database content from an existing replica, if not already available, and (ii) copying and restoring that content on a new replica. These operations can take minutes or hours depending on the database size.

In fact, traditional "just-in-time" cloud provisioning techniques, including Amazon Auto Scaling [1], are similarly based on lazy triggers and/or thresholds and do not take into account the time to replicate the database state. If this state replication and synchronization overhead is ignored, the newly provisioned capacity comes online far too late to handle the workload increase and the capacity requirements will not be met in a timely fashion.

Similarly the "how" to achieve the desired capacity increase must be handled differently in case of dynamic database provisioning. Typically this part involves (i) a capacity determination model to estimate how many replicas to provision for a given workload, and (ii) the actual system steps necessary in starting up and configuring those replicas for use. Capacity determination models predict the future workload using historical data or dynamic predictors [11] and then use queuing techniques to estimate the number of replicas needed to service the predicted workload [9]. This aspect of provisioning is similar for both the front-end web and the back-end database tier. In fact, one of the few papers to address dynamic provisioning of the database tier [8] proposed an analytical model for databases to determine capacity needed to service a given workload. However, this work did not address the important systems issues of "when" to trigger provisioning based on state replication overheads, nor did it address the many system challenges involved in dynamically starting up database replicas. Specifically, in database provisioning, even after a VM replica starts up, there is an additional overhead of synchronizing the state of the new replica with the current state of all other replicas to preserve data integrity. No such overheads are incurred when provisioning "stateless" web and application tier replicas.

Thus, database provisioning differs significantly from traditional web server provisioning because databases are stateful and their state can be very large (and this state must be replicated before a new database replica can be spawned). To provision database replicas in a timely fashion, it is necessary to know how much time will be required to replicate/synchronize this disk state and bring the replicas online. These times vary greatly with the database size, schema complexity, backup/restore tool options, database artifacts (e.g., storage engine configuration). Moreover, there are many tradeoffs on how and when to snapshot the database state to minimize replica resynchronization time. It is therefore non-trivial to estimate the exact time needed to spawn a new replica.

1.2 Research Contributions

In this paper, we present Dolly[1], a system for dynamically provisioning database replicas in cloud platforms. Dolly is database platform-agnostic and uses virtualization-based replication mechanisms for efficiently spawning database replicas.

The key insight in Dolly is to intelligently use VM snapshots and cloning as the basis for dynamic database provisioning. In Dolly, each database replica runs in a separate virtual machine. Instead of relying on the traditional database mechanisms to create a new replica, Dolly clones the entire virtual machine (VM) of an existing replica, including the operating environment, the database engine with all its configuration, settings and the database itself. The cloned VM is started on a new physical server, resulting in a new replica, which then synchronizes state with other replicas prior to processing user requests.

Our work on Dolly has led to the following contributions:

- *When to provision:* Dolly takes the long latency of spawning database replicas into account when triggering "eager" provisioning decisions. To do so, Dolly incorporates a model to estimate the latency to spawn a replica, based on the VM snapshot size and the database resynchronization latency, and uses this model to trigger the replica spawning process well in advance of the anticipated workload increase.
- *How to provision*: Dolly incorporates an intelligent scheduling technique that can determine whether it is cheaper to take a new VM snapshot or use an older snapshot when spawning a new replica. In addition, the technique can proactively trigger VM snapshots to reduce the future latency of spawning database replicas. These mechanisms are implemented in a new provisioning algorithm, with user-defined cost functions to characterize database provisioning policies on cloud platforms. This allows the system administrator to tune the provisioning decisions to optimize resource usage of her cloud infrastructure.
- *Prototype implementation:* We have developed a prototype of Dolly using Sequoia [16], a commercial-grade open-

[1] Inspired by the sheep Dolly, the first mammal to be cloned successfully.

source database clustering middleware, and have combined it with the OpenNebula [14] cloud manager to address provisioning in both private and public clouds. We demonstrate the efficacy of Dolly in provisioning *Mysql*-based database tiers.

- *Evaluation on public and private clouds:* We conduct an experimental evaluation of Dolly on Amazon's EC2 public cloud and on a laboratory-based Xen private cloud. Our experiments with a TPC-W [19] e-commerce workload show the ability of Dolly to properly schedule provisioning decisions to meet capacity requirements in a timely fashion while optimizing resource usage in private clouds and minimizing cost in public clouds.

The remainder of this paper is organized as follows. Section 2 introduces the necessary background on database replication and replica spawning. Section 3 discusses the core techniques for database provisioning in the cloud and *when* to provision, while section 4 addresses *how* to provision. Section 5 presents Dolly's implementation. We perform an experimental evaluation on private and public clouds in Section 6. Finally, Section 7 discusses related work before concluding in Section 8.

2. Background

In this section, we present background on virtualized cloud platforms and database replication and also formulate the problem of dynamic database provisioning.

2.1 Virtualized Cloud Platforms

Our work assumes a virtualized cloud platform that runs distributed web-based applications. The cloud platform is essentially a data center that provides compute and storage resource to its applications. Each physical server in the data center is assumed to run a virtual machine monitor (aka hypervisor) and one or more virtual machines that encapsulate application components. The cloud platform, whether public or private, is assumed to support on-demand allocation of virtual machines—applications can request one or more virtual machines at any time, upon which the requested VMs are created, placed on to physical servers with idle capacity and started up. We assume that application components are preconfigured as virtual disk images that are available to the cloud platform, enabling automated VM allocation and startup. Such on-demand VM allocation is a prerequisite for our dynamic provisioning techniques. Our work targets both public cloud platforms such as Amazon EC2 as well as private Linux-based clouds constructed using Xen/KVM virtualization platforms. In case of public clouds, where servers and storage is charged based on a pay-as-you-use model, we assume that the pricing model is known a priori and can be taken into account when making provisioning decisions.

2.2 Problem Formulation

We assume that cloud platforms run distributed web applications. Each application is assumed to employ a multi-tier architecture consisting of a front-end web tier, a middle application (e.g., J2EE) tier, and a backend database tier. Each tier is assumed to be dynamically replicable. That is, each tier is assumed to comprise one or more VM replicas, and it is assumed that the number of replicas at each tier can be varied based on changing workload demands at that tier. We assume that each tier also assumes a dispatcher/load-balancer that is responsible for distributing requests to various replicas.

Our work focuses on the database tier. In contrast to prior work that has typically assumed a static number of (over-provisioned) replicas at the database tier, we assume that this tier can also be dynamically provisioned like the other tiers. The dynamic provisioning problem for the database tier can then be stated as two sub-tasks: (i) *when* to trigger a capacity change based on current and future workload trends, and (ii) *how* to startup or shutdown the desired number of VM replicas. As discussed above, both tasks raise new challenges when dynamically provisioning databases. Our goal is to design a database provisioning platform that (i) given future workload forecasts, will estimate the time to start up a new database replica and will use this latency to trigger a provisioning change sufficiently in advance of the anticipated change, and (ii) uses an intelligent algorithm that takes user-specified cost functions to optimize the overheads of starting up (or terminating) the desired number of replicas.

2.3 Database Replication

Before presenting our provisioning technique, we present brief background on database replication. Dynamic database provisioning assumes that the underlying database platform is replicable and clusterable. In general, there are two primary architectures for implementing database replication: *shared-disk* and *shared-nothing*. In the *shared-disk* architecture, there is a single copy of the data on a shared disk (SAN or NAS) that is accessed by all replicas. Typically shared-disk database platforms such as Oracle RAC [13] require specific hardware (in the form of shared disk systems) that may not be available in commodity cloud platforms.

In the *shared-nothing* architecture, there are multiple database server processes that run on different machines, and each replica has a copy of the database content on its local disk. Consistency is maintained across replicas using LAN communications. Dolly currently assumes a shared-nothing architecture since they are well suited for today's cloud platforms and also commonly used in multi-tier web applications.

Within shared-nothing systems, there are two main replication strategies: master-slave and multi-master. In *master-slave*, updates are sent to a single master node and lazily replicated to slave nodes. Data on slave nodes might be stale and it is the responsibility of the application to check for data freshness when accessing a slave node. *Multi-master* replication enforces a serializable execution order of transactions between all replicas so that each of them applies update transactions in the same order. This way, any replica can serve any read or write request.

Further, replication can be implemented inside the database engine, also known as *in-core* replication, or externally to the database, commonly called *middleware-based* replication. The technique to add a new replica is similar in both environments. In both architectures, transactions are balanced among the replicas and are stored in a transactional log (also called recovery log). The middleware design usually keeps a separate transactional log for replication, whereas the in-core approach stores the information in each database's replica transactional log.

Figure 1 shows the steps to spawn a replica in a middleware-based replication environment. First, a command to add a new replica is issued from the management console to the replication middleware. A checkpoint is then created in the transactional log (step 2) and a replica is temporarily taken out of the cluster to take a snapshot (also called database dump) of the database content (step 3 via DB_2). As soon as the snapshot has been taken, this replica is resynchronized by replaying the transactions written in the transactional log since the checkpoint (step 4) and it rejoins the cluster. A new replica is then started on a separate node, and the snapshot is seeded to this new replica using a restore operation (step 5). Finally, the updates that have occurred since the snapshot was taken are replayed from the transactional log (step 6) to resynchronize the new replica and bring it up-to-date with all other replicas in the system.

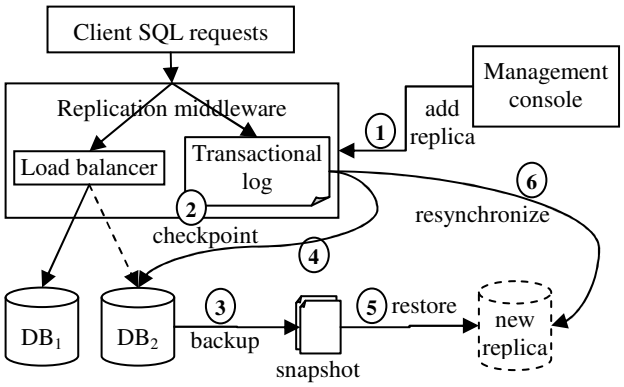

Figure 1. Procedure to spawn a replica.

Conceptually, the above steps for replica creation can be classified into three key phases: (i) the *backup* phase, where database content is extracted from an existing replica and moved to a new node, (ii) the *restore* phase, where a new replica is seeded with this snapshot, and (iii) the *replay* phase, where the replica is resynchronized with others by replaying new updates from the transactional log.

As we will see in the Dolly design, the use of virtualization simplifies these steps. Dolly employs VM snapshots to implement the backup phase and uses VM cloning to restore the snapshot onto a new replica. By doing so, Dolly is *database-platform agnostic*, since it relies on the virtualization platform, rather than native platform-specific tools, to implement provisioning via backup/restore. VM cloning is independent of the database schema complexity and eliminates common backup issues of database specific extensions and configurations [7].

Further, the Dolly design is general and can accommodate both master-slave and multi-master shared-nothing architectures; our current implementation, however, is based on a multi-master middleware-based replication and is implemented on Sequoia, a commercial-grade database clustering middleware [16].

3. Dolly: When to Provision

In this section, we first describe the high-level approach used in Dolly to provision database replicas via VM snapshot and cloning and then present a model to estimate the latency of these operations when provisioning a new replica.

3.1 Replica Spawning via VM Cloning

Dolly uses the ability to make VM snapshots and clone VMs to efficiently replicate database state and start new replicas. Figure 2 illustrates the high-level approach to spawn 2 new replicas in a private cloud. First, the virtual machine (VM) running a database replica is stopped on machine 1 and cloned to be stored on a backup server (machine B). Two new replicas are then spawned by cloning the VM from the backup server and starting these new VMs. Dolly minimizes the downtime of the existing replica that is being cloned by exploiting VM or file-system-level snapshots. A file-system or VM-level snapshot [5] is a point-in-time copy of the virtual disk image; snapshots can be made efficiently, after which the original VM replica can be resumed immediately and the snapshot image can be copied to the other machine(s) in the background.

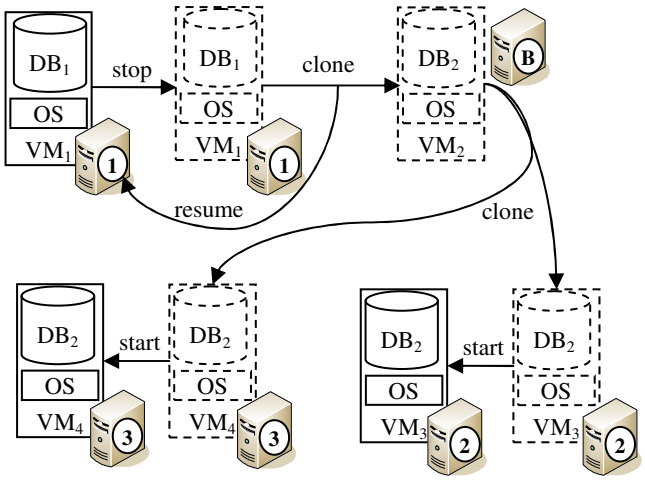

Figure 2. Replica spawning in a private cloud.

Figure 3 shows how spawning 2 replicas works in a public cloud such as Amazon EC2 that provides a Network Attached Storage (NAS) service called Amazon Elastic Block Storage (or EBS). Note that EBS volumes cannot be shared by multiple instances and are therefore different from a SAN or shared disk approach.

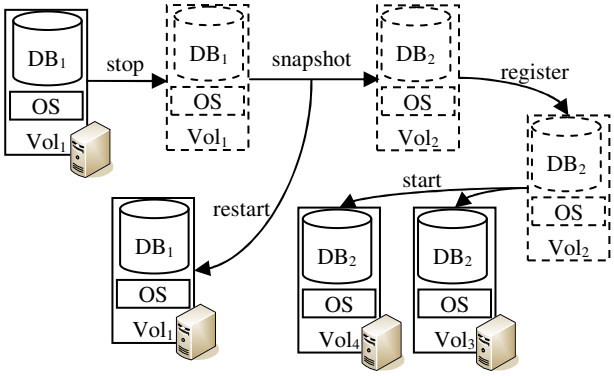

Figure 3. Replica spawning in a public cloud.

The VM disk image is stored on an EBS volume and the VM boots from this image. When the VM is stopped, the volume is detached from its running server. EBS allows snapshots of the volume to be created; doing so asynchronously replicates the volume. The volume snapshot must then be registered in EC2 in order to create new VMs. This is equivalent to storing the image

to a backup server in a private cloud. When a new VM is created from an EBS snapshot, a clone of that volume is created and dedicated to the newly started instance. In our case, we assume that the database server disk state (configuration file and data within the database) are stored on the EBS volume; thus snapshots and booting a new VM from the snapshot is an effective mechanism to replicate the shared-nothing database content and start up a new database replica.

3.2 Determining replica spawning time

Dolly must accurately estimate the overheads of the above VM snapshot and cloning operations in order to intelligently trigger the spawning of new replica(s). We now present a simple model to estimate this latency.

In general, there is a tradeoff between the time to snapshot/clone a database/VM, the size of the transactional log and the amount of update transactions in the workload. For example, a new replica can be seeded with an old snapshot (e.g., a snapshot that was taken to seed a different replica), which eliminates the backup phase overhead. However, use of an older snapshot forces the system to keep a larger transactional log and also increases the time to replay updates from this log during the *replay* phase. On the other hand, taking a new snapshot for each new replica may incur significant overheads during the *backup* phase, especially if the database is large. By analyzing the overheads of these operations, Dolly can choose the option with the lower latency.

The replica spawning overhead can be analyzed using the five variables defined in Table 1.

Table 1. Replica spawning time variables.

b_i	backup time to generate VM snapshot *i*
r_i	time to restore/clone snapshot *i* on a new replica
$replay_i$	time to replay update transactions logged since snapshot *i*
w_t	average update transaction throughput observed at the time the new replica spawning command is issued
w_{max}	maximum update transaction throughput of the replica

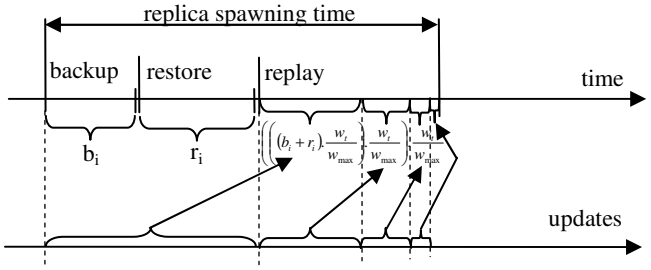

Figure 4. Decomposition of the replica spawning time with a new snapshot.

When no snapshot is available, it is necessary to perform a new backup and restore, yielding an overhead of (b_i+r_i) as shown on Figure 4. The replay phase then replays all updates that have occurred during this period. We can estimate the replay time by observing the current rate of update transactions and assume that it will remain a valid approximation during the replay time. The new replica will be able to replay the requests at w_{max} speed since it does not have to execute any other transaction. Therefore, the time to replay the updates that occurred during backup/restore is $(b_i + r_i) w_t / w_{\max}$. Since new updates will occur during this replay, it will take an additional $\left((b_i + r_i) w_t / w_{\max}\right) w_t / w_{\max}$ to replay them. This is the geometric series with: $p = \frac{w_t}{w_{max}}, w_t < w_{max}$, $\sum_{k=0}^{\infty} p^i = \frac{1}{1-p}$

If the system is under peak load, $w_t = w_{max}$, the replica will never be able to catch up and it will have a lag of b_i+r_i.

Table 2. Replica spawning time formulas

Replica spawning time when no snapshot is available	$(b_i + r_i)\frac{w_{\max}}{w_{\max} - w_t}$
Replica spawning time from an existing snapshot *i*	$(r_i + replay_i)\frac{w_{\max}}{w_{\max} - w_t}$

We find the equations in Table 2 and conclude that: *it is faster to take a new snapshot j to spawn a new replica if* b_j+r_j < r_i+$replay_i$. Any dynamic provisioning technique for replicating the database tier of the application needs to consider this key tradeoff. The VM cloning mechanism used by Dolly provides a predictable backup/restore time independent of the database size and schema complexity as shown in Table 8. Cloning only depends on the VM image size that is known and its snapshotting time can be easily predicted. $replay_i$ can be accurately predicted by recording the execution times of each update transaction and adding them up.

Since $replay_i$ can be accurately predicted, having a constant b_j and r_j, that are independent of the database size or complexity, allows Dolly to decide if b_j<$replay_i$ in which case it is faster to take a new snapshot than to use an existing one to spawn a new replica.

4. Dolly: How to Provision

Our Dolly provisioning system has four main components: capacity provisioning engine, snapshot scheduler, paused pool cleaner and actuator. Typically, the provisioning engine will employ a workload predictor (Section 4.1) that observes the behavior of the system. To provision a certain capacity by a given deadline, it is necessary to schedule *capacity provisioning* actions according to the time it takes to replicate the database state (Section 4.2). As replicas have to be spawned from a database snapshot, the *snapshot scheduler* decides when new database snapshots (VM clones) have to be taken (Section 4.3). Some stopped or paused VMs become obsolete over time and need to be purged by the *paused pool cleaner* (Section 4.4). The *actuator* orchestrates and executes the orders of all the other components.

Whenever new workload predictions become available, the capacity provisioning algorithm is invoked to compute a new schedule to meet capacity demands. Then the snapshot scheduler runs to check if new snapshots could be generated (possibly from paused VMs) to make future spawning operations cheaper. If new VM snapshots are generated, we re-run the capacity provisioning algorithm to generate a new schedule. In the end, we obtain a schedule of snapshot and capacity provisioning actions (adding, pausing, resuming replicas) that are executed by the actuator. Dolly also regularly triggers the paused pool cleaner to free old paused VMs and snapshots that are no longer needed. A more detailed description of the algorithms is available in [6].

To adapt provisioning policies to the target cloud platform, Dolly uses cost functions to allow the administrator to define which option is best if multiple strategies are available. The cost can model any metric like time, resource usage or actual resource cost as we will show in the next sections. Table 3 lists the seven cost functions used by Dolly and the definitions for each.

Table 3. Cloud platform specific cost functions used by Dolly.

Cost function name	Definition
`pause_cost(VM, t)`	cost of pausing VM at time t
`spawn_cost(s, t, d)`	cost to spawn a replica from snapshot s at time t to meet deadline d
`spawn_cost(VM, t, d)`	cost to spawn a replica from a paused VM at time t to meet deadline d
`running_cost(VM,t1,t2)`	cost to run a VM from time t1 to time t2
`pause_resume_cost(VM, t1, t2)`	cost to pause a VM at time t1 and resume it at time t2
`backup_paused_cost(VM)`	cost to backup a paused VM
`backup_live_cost(VM, t)`	cost to backup an active VM at time t

Table 4 summarizes the variables used to measure the time used by the different operations used by the algorithms described in this section.

Table 4. Variables used to measure replica spawning operations**.**

rr	Time to restore and replay from the latest snapshot
br	Time to spawn from a new snapshot (backup+restore)
rs_{VM_i}	Time to resume paused VM *i*
psr	Time to pause/snapshot/resume a VM
pw	Prediction window

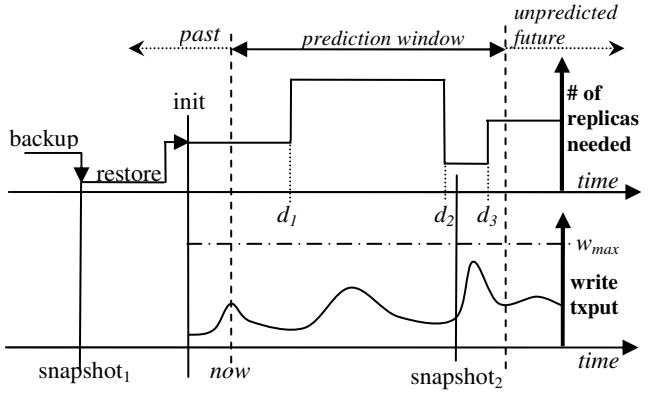

Figure 5. Example of a capacity and write workload prediction over time. Dolly provision replicas based on the forecast available in the prediction window.

4.1 Capacity and workload predictors

Previous work has established how to predict replicated database capacity based on a standalone node measurement [9]. This allows forecasting performance scalability and identifying potential bottlenecks. Many models exist for workload prediction [11], [20]. Dolly does not assume any particular workload predictor or capacity model; it can use any existing approach and can be a platform to test new predictors or improve existing ones.

Depending on the capacity and workload predictors used, the forecast will have a limited visibility in the future. Web sites with stable workloads might have accurate static weekly predictions possibly adjusted by administrators for seasonal peaks. More dynamically changing workloads can be less predictable and only sketch the demand for the next hour or so. We call *prediction window* the time between now and the latest time in the future for which the load and capacity demand can be predicted.

Figure 5 shows an example of capacity demand and write throughput of a replicated database. The prediction window slides as time goes on. Prediction windows are not necessarily of a fixed size since a predictor can dynamically change the technique it uses to forecast the load thus increasing or decreasing the prediction window size. Dolly has to schedule provisioning decisions for deadlines d_1, d_2 and d_3, where the capacity demand changes in the prediction window.

4.2 Provisioning replicas

The provisioning algorithm scans the prediction window and looks for deadlines where changes in workload require additional capacity (such as time d_1 and d_3 on Figure 5) or less capacity (such as time d_2 on Figure 5). The algorithm handles all deadlines in sequence. In Figure 5, d_1 is handled first. Once a schedule has been found for d_1, it moves to d_2 and so on. The algorithm works in two phases for each deadline: 1) list all possible options for replica spawning or releasing and 2) sort these options according to a cost function.

4.2.1 Decreasing capacity

When the capacity requirements decrease, replicas that are no longer needed are paused. The replication engine keeps track of the state of each stopped virtual machine replica so that it knows exactly what has to be replayed when the VM is resumed. A similar state is saved in the slave nodes for master/slave replication.

When a VM is stopped in a private cloud, its image still resides on the machine's local disk. As we might want to resume that image later, we do not return the machine to the free server pool but it is put it in a special *paused server pool*. The machine can be shutdown as long as it is in the paused pool. A machine can be reclaimed from the paused server pool by the private cloud infrastructure if the free pool is empty and additional capacity is required for other databases or tiers. In a public cloud like EC2, the computing instance is simply detached from the storage and can be re-attached later to any other instance.

The platform specific cost function, `pause_cost(VM, d)` determines the cost of pausing *VM* at time *d*. For example, in EC2 where server time is billed by the hour, if at time *d* VM_1 has just started a new billed hour and VM_2 is toward the end of its billed hour, we would have pause_cost(VM_1, d)>pause_cost(VM_2, d). On a private cloud, the administrator might prefer to switch off the hottest machines to improve cooling. If the capacity has to be reduced by *r* replicas at time *d*, the algorithm schedules the *r* replicas that have the lowest `pause_cost` for pausing.

4.2.2 Increasing capacity

When an increase in capacity is predicted at deadline d, the algorithm explores all replica spawning options from snapshots and paused VMs.

In our system, the replicated database always has at least one snapshot available for creating new replicas. The first snapshot is created when initializing the system as shown on Figure 5, and snapshots are updated regularly when needed, as will be explained in section 4.3. When new replicas are spawned from a snapshot,

we can predict the time it takes to bring the replica online using the formula described in section 3.2.

Dolly looks at all available snapshots that can spawn replicas in time to meet deadline d, as well as all paused VMs that can be resumed and resynchronized in time. Each option has its own cost defined by the `spawn_cost` function. For example, on a private cloud, options using the latest start times allow unused nodes to remain switched off longer and save energy. On a public cloud such as EC2, the cost can be defined by the price the user is going to pay for the compute hours of the instance, the IOs on EBS and the monthly cost for data storage.

The cheapest options are selected to be executed. Note that if there are not enough options to provision all replicas, this means that it is not possible to spawn all replicas in time for the deadline given the current workload. We address this scenario in the next section.

4.2.3 Admission control

If a capacity deadline cannot be met in time with the current forecast, it is possible to perform admission control on the system in multiple ways. Note that this scenario can only happen if the predictor drastically changes its predictions for the current prediction window (such as an unpredicted flash crowd).

First we assume that no writes will update the system from now on and compute the time it takes to restore and replay from the latest snapshot (*rr*), to take a new snapshot and spawn a replica from it (*br*=backup+restore) or resume from paused VMs (rs_{VM_i}).

If we find that $now + min(rr, br, rs_{VM_i}, rs_{VM_j}) \le d$, this implies that there is enough time to create replicas but the write throughput is too high or too close to w_{max} for replicas to catch up in time. Doing admission control on the write throughput w_t can be used to meet the deadline as long as:

$$w_t \le w_{\max} - \frac{min(rr, br, rs_{VM_i}, rs_{VM_j}).w_{\max}}{d - now}$$

Note that doing admission control on writes (*write throttling*), means that update transactions are going to be delayed. Depending on timeout settings, this might translate into transactions being aborted. The minimum acceptable write throughput can be set by the administrator.

If replicas cannot be spawned in time even with write throttling, it is necessary to perform admission control on the incoming workload to prevent the system from crashing due to overload. Admission control can be performed by the replication engine by allowing only a fixed number of transactions in the system at any given time. It can also be achieved at another tier in front of the database (e.g. web tier admission control). A workload matching the current capacity has to be maintained until additional capacity becomes available at time:

$$d - now + min(rr, br, rs_{VM_i}, rs_{VM_j}) \frac{w_{\max}}{w_{\max} - w_t}$$

The administrator can set a minimum acceptable w_t and let Dolly perform admission control and schedule spawning operations accordingly.

4.3 Scheduling new database snapshots

In addition to provisioning new replicas or pausing existing ones, Dolly must deal with the problem of periodically creating new database snapshots by cloning VMs. A newer snapshot reduces the cost of spawning a new replica in the future (since it has a more recent version of the database and will incur a lower synchronization overhead). However, creating a snapshot incurs an overhead, and Dolly must intelligently schedule their creation to balance the cost and the benefit.

Two problems have to be solved to schedule new database snapshots: *how* and *when*. *How* can either be from an already paused VM or by pausing an active VM for the time of the snapshot (see section 4.3.1). A new snapshot must be ready *when* the time to restore and replay from the previously available snapshot is greater than the prediction window (see section 4.3.2).

4.3.1 How to snapshot?

An opportunistic method to create a new snapshot is to clone VMs that have been paused. While a paused VM only captures the database state until the time it was paused, it might still be a significant improvement over the last snapshot available.

The only other option requires taking an existing replica offline for the time of the pause/snapshot/resume (psr) operation and replaying of updates that happened since the VM was paused. This means that the capacity of the system is going to be reduced by 1 replica from t_{backup} to $t_{backup} + \left(psr + replay_{t_{backup}} \right) \frac{w_{\max}}{w_{\max} - w_t}$.

If the workload prediction does not allow a replica to be temporarily disabled during that time interval, an additional replica has to be provisioned at time t_{backup} to allow taking a new snapshot. This new deadline can be added to the current capacity prediction and the capacity provisioning algorithm described in section 4.2 has to be re-executed to provision this additional replica in time.

4.3.2 When to snapshot?

If we want to provision additional replicas in time, the time to restore and replay from the latest available snapshot should never exceed the prediction window. Otherwise, when the predictor forecasts a new capacity demand increase at the end of the prediction window, there would not be enough time to spawn new replicas. This means that a new snapshot must be ready to be fully restored at time t_{switch} defined by: $r_{backup_i} + replay_{backup_i,switch} = pw$

where *pw* is the prediction window and $replay_{backup_i,switch} = \sum_{t=t_{backup_i}}^{t_{switch}} \frac{w_t}{w_{\max}}$

To make sure that additional replicas can be provisioned at t_{switch} using the new snapshot, the backup operation must be started prior to time $t_{backup_{i+1}}$ so that there is enough time to backup, restore and replay a new replica at time t_{switch}. This translates to:

$$b_{backup_{i+1}} + r_{backup_{i+1}} + replay_{backup_{i+1},switch} \le t_{switch} - t_{backup_{i+1}}$$

To guarantee that a new snapshot can be ready in time, the prediction window must be long enough so that:

$$pw \ge t_{switch} - t_{backup_{i+1}} \ge b_{backup_{i+1}} + r_{backup_{i+1}} + replay_{backup_{i+1},switch}$$

If the prediction window is too short or write throughput is too high, admission control can be used to make sure that new snapshots can be prepared in time within the prediction window.

The algorithm then scans the prediction window and look at each deadline where new replicas have to spawned (adding capacity

only). For each deadline, it calculates the cost to spawn new replicas for 3 strategies:

1) The cost to spawn replicas from a snapshot given by `spawn_cost` (defined in section 4.2.2) for all snapshots that can be restored and replayed by the deadline.

2) For each paused VM (step 3) that can be snapshotted, restored and replayed by the deadline, the cost to take the backup from the paused VM is given by the cost function `backup_paused_cost` to which we add the cost of spawning replicas from this backup.

3) The cost of creating a backup from a live replica is given by the `backup_live_cost` function to which we add the cost of spawning replicas from this backup and the eventual cost of bringing a replica online if no idle replica is available.

Next, the algorithm keeps the option that has the minimal cost for each deadline and schedules the operations accordingly. If no option is available to spawn a replica in time for a given deadline, the algorithm computes at what time a snapshot should be taken and modifies the capacity requirements to ask for one replica to be ready by that time. The capacity provisioning is then invoked to provision that replica, eventually using admission control if needed.

The capacity provisioning algorithm is re-run every time new snapshots have been scheduled to check if a better replica spawning schedule is available. If this is the case, the old schedule is replaced by the new schedule.

4.4 Relinquishing resources

Over time, some paused VMs become obsolete and are not cost effective to be resumed. The same applies to old VM snapshots that need to be erased. The *paused pool cleaner* has the responsibility of releasing these resources. It is invoked at regular time intervals that can be set by the administrator (from every hour, to every day or every week). It scans each paused VM and checks the cost of resuming that VM (*spawn_cost(VM, now, pw_{end})*) and compares it to the cost of spawning a replica from the latest available snapshot (*spawn_cost(b_i, now, pw_{end})*). If the cost of resuming the VM is higher, it means that this VM will not be used anymore and it can be released.

A similar approach can be used for snapshots. All snapshots that are older than the current latest available snapshot can be released. However, the administrator might want to keep multiple older backups for recovery purposes. On a public cloud like EC2, since storage is paid for on a monthly basis, a better policy may be to retain old volumes until the end of the billing cycle.

4.5 Current limitations

Dolly assumes that all the components of the cloning operation (backup, restore, snapshot…) have a constant time which is correct for homogeneous setups with LAN interconnections. This might not be the case with heterogeneous resources or resources in different EC2 regions or clouds using WAN interconnections. The worst case scenario measurement could be taken to ensure safe scheduling, but specific optimizations for such environments are left to future work. Additional optimizations such as virtual machine migration can also be considered in these environments.

When synchronizing slave nodes in a master/slave setup, the synchronization process uses master node resources and potentially impacts its performance. We have not currently modeled this performance impact but we did not find it noticeable in our early experiments.

5. Dolly Implementation

We have implemented the concepts of Dolly in the Sequoia 4.0 [16] database clustering middleware and integrated it with the OpenNebula cloud infrastructure manager v1.4 [14]. OpenNebula works with both private and public cloud resources and offers a single API to manipulate VMs independently of the target platform. Figure 6 shows an overview of the integration of Dolly with Sequoia and OpenNebula in the context of the TPC-W benchmark.

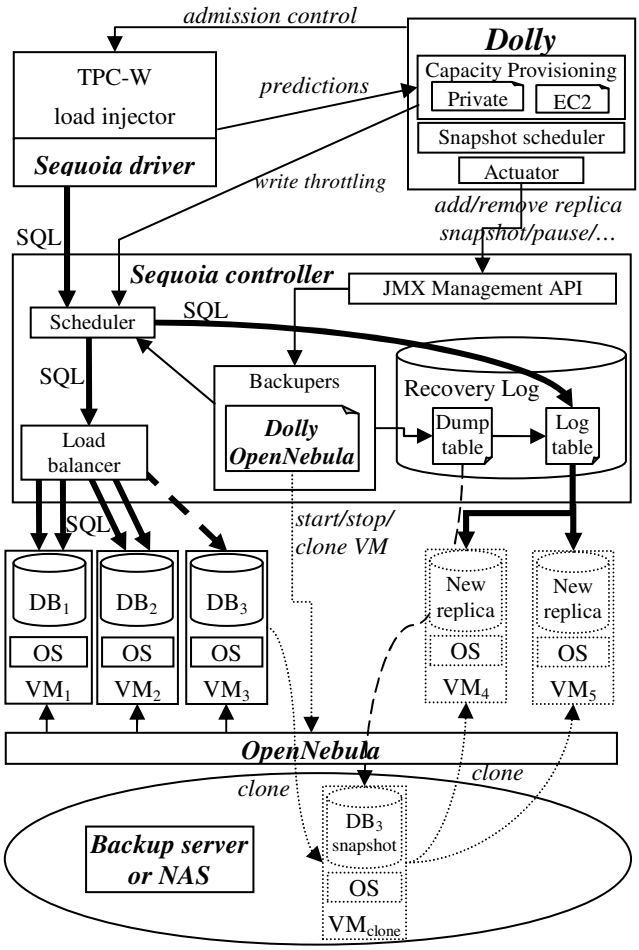

Figure 6. Overview of Dolly integration in Sequoia and OpenNebula running the TPC-W benchmark.

Client applications send SQL requests to the Sequoia controller that forwards them to the underlying databases to perform replication. The SQL commands of update transactions are recorded with their execution time in a transactional log called *recovery log*. The log itself is stored in an embedded database running within the Sequoia controller. The recovery log can be replayed to synchronize new or failed replicas. Additionally, Sequoia has a replica spawning infrastructure with a pluggable *backuper* interface that interacts with the recovery log and allows for database specific implementations of backup and restore operations. We have implemented a Dolly/OpenNebula backuper that interacts with OpenNebula to start/stop and clone/snapshot

virtual machines to implement the backup and restore functionality. When a new backup is triggered, a pointer to the current state of the recovery log is stored with the dump metadata. When a restore operation is launched, the dump is first restored and dedicated threads then replay the recovery log (i.e. re-execute the SQL commands) from the point that was saved in the metadata. Updates are applied in a serializable order to bring the new replica in a consistent state with other replicas. The time to replay is computed by summing the recorded execution time of all queries to replay. More information about Sequoia internals and its recovery log can be found in the Sequoia documentation [16].

Dolly takes predictions directly from the TPC-W load injectors that act as oracles with perfect information. A tunable prediction window can be used from 1 minute to the entire length of the benchmark run. The provisioning actions are directly sent to the Sequoia controller through its administration interface. Dolly performs admission control directly on the load injectors but it would typically do this at the web tier level in a multi-tier setup. The write throttling is achieved by interacting with the Sequoia scheduler. We have implemented different cost functions to model our private cloud platform and the Amazon EC2 public cloud.

The private cloud cost functions detailed in pseudo-code in Table 5 optimize the time the resources are used. The longer the resources are used, the more power they use and the higher the cost. When the algorithm has to decide which VM to pause, it selects the hottest machine at that time.

Table 5. Cost function implementation for our private cloud.

Cost function name	Implementation
`pause_cost(VM, t)`	`return 1/VM->machine->temp`
`spawn_cost(s, t, d)`	`return d-t`
`spawn_cost(VM, t, d)`	`return d-t`
`running_cost(VM,t1,t2)`	`return 1`
`pause_resume_cost(VM, t1, t2)`	`if (t2-t1>VM->pause+VM->resume) return 0 else return 2`
`backup_paused_cost(VM)`	`return backup_time`
`backup_live_cost(VM, t)`	`return VM->pause + backup_time + VM->resume`

Table 6 models the cost functions as the real cost the user would pay for EC2 resource usage. It includes both the compute time for server instances (charged by the hour at the *hour$* rate) and the IO cost (charged monthly per GB of storage (*EBS_storage$*) and IOs are charged per million (*EBS_io$*)). EBS snapshots are stored on S3 and are charged monthly per GB of storage (*S3_storage$*).

Table 6. Cost function implementation for Amazon EC2.

Cost function name	Implementation
`pause_cost(VM, t)`	`return 60-((t-VM->start)%60)`
`spawn_cost(s, t, d)`	`comp$=(d-t)/60*hour$ io$=EBS_storage$*s->size + EBS_io$* (s->restore_io+s->replay_io) return comp$+io$`
`spawn_cost(VM, t, d)`	`comp$=(d-t)/60*hour$ io$= EBS_io$* (s->resume_io+s->replay_io) return comp$+io$`
`running_cost(VM,t1,t2)`	`(t2-t1)/60*hour$;`
`pause_resume_cost(VM, t1, t2)`	`io$= EBS_io$* (VM->pause_io+VM->resume_io) comp$=(60-(VM->stop-VM->start) %60)/60*hour$ return io$+ comp$`
`backup_paused_cost(VM)`	`return S3_storage$*s->size`
`backup_live_cost(VM, t)`	`return pause_cost(VM, t)$+ S3_storage$*s->size + (VM->stop_io+VM->start_io)* EBS_io$`

6. Experimental Evaluation

This section first introduces the cloud platforms used for our experiments. We then present our performance evaluation.

6.1 Cloud Platforms

We conduct experiments on private and public clouds. We use a private cloud composed of a cluster of Pentium 4 2.8GHz machines. Each machine is running a CentOS 5.4 Linux distribution with a Linux kernel version 2.6.18-128.1.10.el5xen, the Xen 3.3 hypervisor, Java runtime version 1.6.0_04-b12and MySQL v5.0.45. All machines are interconnected by a Gigabit Ethernet network.

We use Amazon EC2 as our public cloud. EC2 instances are created from EBS volumes. We use standard large on-demand EC2 instances in our experiments. Each EC2 instance has CloudWatch running on it to monitor the number of writes. The price of our EC2 instance with CloudWatch is $0.355 per hour. The price of an EBS volume is $0.10 per allocated GB of data per month. The cost of doing I/O requests to an EBS volume is $0.10 per million I/O requests. There is a cost of $0.15 per GB per month associated with the storage of EBS volume snapshots.

Table 7. Operation timings in seconds for a TPC-W benchmark virtual machine on our private cloud and EC2.

Operation	Private Cloud	Public Cloud (EC2)
start VM	42s	220s
pause VM	26s	30s
resume VM	42s	30s
backup (stop/clone)	150s	320s
restore (clone/start)	165s	220s
w_{max}	149 writes/sec	197 writes/sec
Avg IOs per write	15	13

We build a 4GB VM image of the TPC-W benchmark for both cloud platforms. We report our measurements of the various VM management and cloning operations in Table 7. We measure the maximum write throughput of a single replica (w_{max}) obtained by running only write transactions of the TPC-W workload on a standalone database. The average number of IOs per write transaction is calculated by running iostat before and after the w_{max} run.

6.2 VM Cloning vs Database Backup/Restore

VM cloning is an alternative mechanism for replicating content when compared to the traditional database-specific backup-restore mechanism. In this section, we compare the copy overheads of the two approaches.

Table 8 shows the time to copy various databases using the database native backup/restore tool (e.g. mysqldump, pg_dump) versus VM cloning. The RUBiS benchmark database [3] is tested with 3 configurations on MySQL using the InnoDB engine: without constraint or index (-c-i), with integrity constraints and basic indexes (+c+bi) and with constraints and full text indexes (+c+fi). TPC-W and TPC-H [19] databases are stored in a PostgreSQL RDBMS. We also experiment with two virtual machine image sizes (4 and 16GB) where we store both the operating system and the database within its content.

Indexes significantly increase the database footprint on disk. We observe from the RUBiS results that integrity constraints checks as well as index building can increase database backup/restore

time by a factor of more than 7 for the exact same database content. Not only do the database schema and backup tool configurations affect timings, different database engines yield very different results for databases with a similar size on disk as shown on Figure 7. We observe that large or complex databases can take more than 1 hour to replicate.

Table 8. Backup/restore and VM cloning time in seconds for various standard benchmark databases.

Database	DB size on disk	DB Backup Restore	Dolly 4GB VM cloning	Dolly 16GB VM cloning
RUBiS –c–i	1022MB	843s	281s	899s
RUBiS +c+bi	1.4GB	5761s	282s	900s
RUBiS +c+fi	1.5GB	6017s	280s	900s
TPC-W	684MB	288s	275s	905s
TPC-H 1GB	1.8GB	1477s	271s	918s
TPC-H 10GB	12GB	5573s	n/a	911s

In contrast, VM cloning performs a filesystem level copy without interpreting database objects, thus it offers a constant time regardless of the database complexity or engine used. The time only depends on the VM image size on disk (280s for a 4GB image and about 900s for a 16GB image). Consequently, since the VM disk size is fixed a priori, VM cloning makes it easy to predict database backup/restore time incurred when spawning a new replica—a crucial pre-requisite for database provisioning. Additionally VM cloning captures the entire OS/database configuration and settings preventing any error in reproducing these settings on the new replica machine.

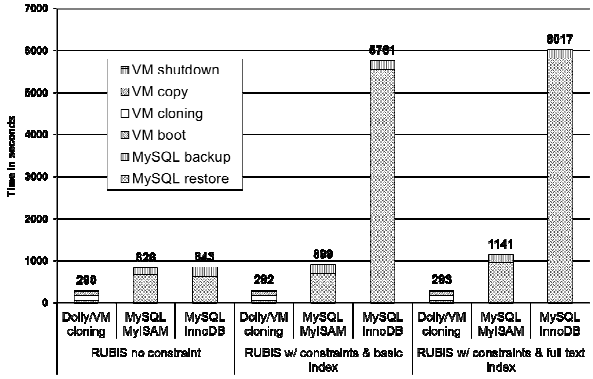

Figure 7. Time breakdown for cloning a database with Dolly and MySQL backup/restore tools with the MyISAM and InnoDB engines using the RUBiS benchmark database with various constraints and indices.

6.3 Provisioning Evaluation

We experiment with TPC-W, an eCommerce benchmark from the Transaction Processing Council [19] that emulates an online bookstore. We use the ObjectWeb implementation of the TPC-W benchmark [17]. The setup is similar to the one depicted in Figure 6 with load injectors providing a 2 hour prediction window. The web tier (not shown on Figure 6) is statically provisioned with enough servers for the length of the experiment.

We compare the provisioning decisions of Dolly for the private and public clouds with traditional provisioning techniques given the workload and initial conditions defined in section 6.3.1.

6.3.1 Workload Description

We have generated a custom mix of interactions to create the workload depicted at the top of Figure 8. We generate a read-only request mix by using the TPC-W browsing mix workload and removing its few write interactions. We use httperf to create the desired number of clients that send these read-only interactions. The write interactions are generated using the customer registration servlet of TPCW. Another set of httperf clients generate these write-only interactions.

We use the model described in [9] to determine the capacity requirements shown in Figure 8. The initial capacity demand at t=0 is 4 replicas (middle graph) and the write throughput is 20% of the maximum write throughput (bottom graph). A snapshot s_0 is also available at time t_0. After 10 minutes the number of replicas needed decreases from 4 to 3. We denote this deadline by d_1. The number of replicas needed decreases further from 3 to 2 at d_2=20 minutes. The capacity demand increases sharply from 2 to 5 replicas at d_3=80 minutes, then drops to 2 at d_4=90 minutes and increases up to 6 replicas at d_5=100 minutes. The number of writes remains constant to 0.2 times the maximum write throughput for one hour with a 10 minute read-only workload starting at d_2. After that hour, the write throughput is 0 until d_3 with a write surge at 50% of the maximum write throughput. The write peak continues for 10 minutes and the write throughput drops to 0 at d_4.

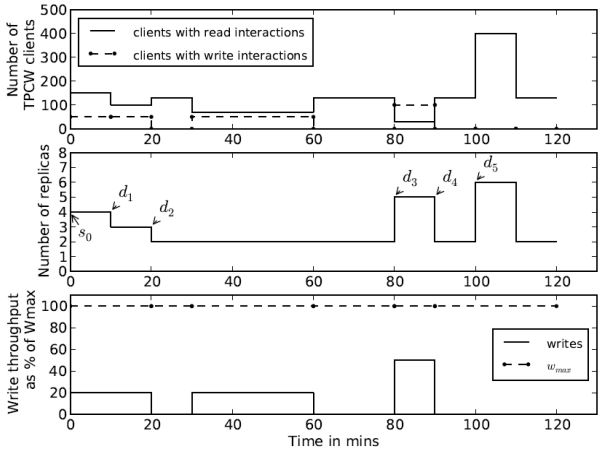

Figure 8. TPC-W workload, predicted capacity requirements and write workload.

6.3.2 Provisioning results

We compare Dolly's performance with two traditional provisioning techniques: *reactive provisioning* and *overprovisioning*. These techniques behave similarly on the private and public clouds.

Reactive provisioning does not use any prediction and just reacts to the current capacity demand. When reactive provisioning is used, database snapshots are generated at fixed time intervals. We use intervals of 15 minutes (Reactive15m), 1 hour (Reactive1h) and 2 hours (Reactive2h), generating 7, 1 and 0 snapshots respectively during the experiment.

The overprovisioning configuration (Overpro6) uses a constant set of 6 nodes. As for reactive provisioning, snapshots are generated periodically. We choose to only generate 1 snapshot during the experiment.

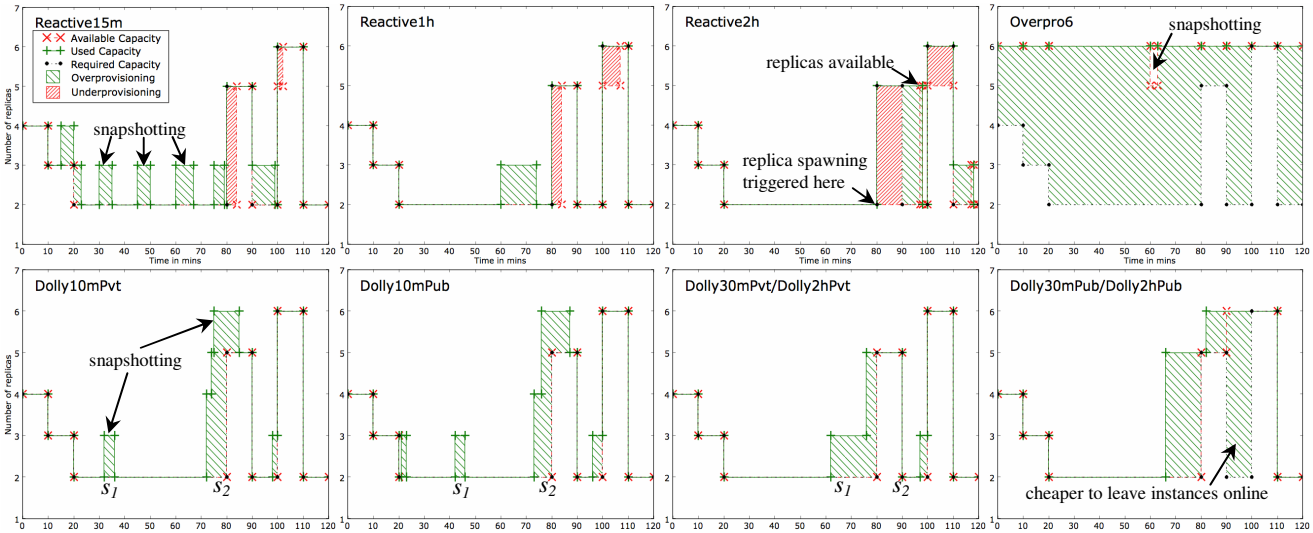

Figure 9. Capacity made available by each provisioning algorithm compared to the required capacity and the total capacity actually used.

We run Dolly with three prediction windows of 10 minutes, 30 minutes and 2 hours. Dolly uses the cost functions presented in section 5 for the private and public clouds. The performance of the different algorithms is summarized in Table 9. The cost for the private cloud represents the cumulative machine uptime (6 machines up for 5 minutes accounts for 30 minutes). The cost for the public cloud (Amazon EC2) is the real cost in $USD. The second metric used is *missing replica minute* (MRM) that measures capacity underprovisioning (i.e. SLA violations). 1 MRM corresponds to a missing capacity of 1 replica for 1 minute (5 replicas missing for 2 minutes accounts for 10MRM).

Table 9. Provisioning algorithm performance for private and public clouds in terms of cost and missing replica minute (MRM).

Provisioning algorithm	Private Cloud		Public Cloud (EC2)	
	Cost (time)	MRM	Cost ($)	MRM
Reactive15m	381m42s	17.5	18.29	27.2
Reactive1h	360m30s	25.8	5.00	33.7
Reactive2h	410m	42.1	4.61	41.5
Overpro6	720m	0	8.39	0
Dolly10m	381m54s	0	7.16	0
Dolly30m	352m	0	3.73	0
Dolly2h	352m	0	3.73	0

The results show that reactive provisioning is not able to properly provision the system with missing capacity ranging from 23.2 to 44.2 missing replica minute. Snapshotting more often reduces the time to spawn new replicas by restore and replay but capacity is missing during the spawning operations.

Overprovisioning (Overpro6) always provides an adequate capacity but at a significantly larger cost on each cloud platform. In contrast, Dolly uses much less resources while still providing the required capacity. A 10 minute prediction window (Dolly10m) requires more snapshots to be able to react to any new capacity demand at the end of the short prediction window. A 30 minute prediction window (Dolly30m) is enough to provide an optimal provisioning using less than half of the resources of the overprovisioned configuration.

Figure 9 shows in more detail the behavior of each algorithm. When reactive provisioning is used, additional capacity is used to spawn a new replica from the latest snapshot so that a new snapshot can be generated. When capacity needs to be increased, the system remains underprovisioned during the time replicas are spawned. The older the snapshot the longer it takes to spawn new replicas. In the Reactive2h case, replicas spawning starting at t=80 completes only 17 minutes later, leaving the system with only 2 available replicas to serve requests during the first peak period.

The Overpro6 configuration constantly provides 6 replicas except for when the snapshot is generated where a node is briefly paused. The large shaded area shows the amount of wasted resources.

Dolly with a 10 minute prediction window (Dolly10m) behaves similarly on both cloud platforms. As the prevision window slides the time to restore and replay from the latest snapshot exceeds the prediction window size. This is why Dolly spawns new replicas to generate new snapshots at deadlines s_1 and s_2. While new replicas are spawned from s_1 during the first capacity increase, the write spike quickly triggers an additional replica to generate s_2. Four replicas are paused at the end of the first peak and resumed for the second peak (no replay time since no write occurred during that paused time). An additional replica is quickly spawned from s_2.

With a 30 minute or longer prevision window (Dolly30m and Dolly2h), decisions change between the private and the public cloud according to the cost functions. While less machine time is used on the private cloud by generating new snapshots from an additional replica online (s_1) or from a paused replica (s_2), the storage cost of a new snapshot dominates the IO cost of replay for EC2. Therefore all replicas are always spawned from the original s_0 snapshot in the public cloud. Instances are also not stopped between the two peaks as instances are paid for a full hour, pausing and restarting them 10 minutes later costs more than letting them run.

In summary, we have shown that Dolly with a prediction window as short as 30 minutes is able to provide optimal resource utilization (according to administrator defined cost functions) while always providing the required capacity.

7. Related Work

Much of the prior work on dynamic provisioning [20], [21], [22], [4] has focused on dynamic provisioning of the front tiers of web applications. In this work we focus on the database tier that differs from other tiers due to its large dynamic state. Commercial solutions such as Oracle RAC [13] use a shared disk approach to avoid the state replication problem. The use of in-memory databases on top of a shared storage has also been considered [12]. Our work focuses on cloud environments where a shared disk approach cannot typically be deployed.

Amazon Relational Database Service (RDS) [2] works with Amazon Auto Scaling [1] to provide reactive provisioning of asynchronously replicated (i.e. master/slave) MySQL databases based on static thresholds. Microsoft in its Azure PaaS (Platform as a Service) cloud offering provides built-in replication in the lower layer of its platform but hides it to the user [15]. Provisioning could be enhanced on both platform using Dolly.

The few papers related to dynamic provisioning of databases usually focus on workload prediction without modeling the time to spawn new replicas [8]. Dolly can work with any load predictor and provisions database replicas accordingly by predicting VM cloning and replica resynchronization time. The problem of re-synchronizing database replicas in a shared nothing environment has been described in [17]. However, the proposed technique only relies on log replay and does not exploit snapshotting as a way to bring up new replicas. Even in a more recent work [10], state synchronization time is based on fixed estimates for replay. We have shown that using virtualization, we are able to snapshot databases via VM cloning and predict state replication time accurately.

8. Conclusion

Database provisioning is a challenging problem due to the need to replicate and synchronize disk state. Since modern data centers and cloud platforms employ a virtualized architecture, we proposed a new database replica spawning technique that leverages virtual machine cloning. We argued that VM cloning offers a replication time that depends solely on the VM disk size and is independent of the database size, schema complexity and database engine. We proposed models to accurately estimate replica spawning time and analyzed the tradeoffs between capacity provisioning and database state snapshotting. To the best of our knowledge, Dolly is the first database provisioning system that can be adapted to the specifics of various cloud platforms via administrator-defined cost functions.

We implemented Dolly and integrated it with a commercial-grade open source database clustering middleware. We proposed different cost functions to optimize resource usage in a private cloud and to minimize cost for the Amazon EC2 public cloud. We evaluated our prototype with a TPC-W e-commerce workload and demonstrated the benefits of an automated database provisioning system for the cloud, with optimized solutions adapted to different cloud platform specifics. We plan to release Dolly as open source software and hope that it will facilitate replicated database deployments in virtualized environments such as clouds.

Acknowledgement

We would like to thank Steve Dropsho for early contributions to this work. This research was supported in part by NSF grants CNS-0834243, CNS-0720616, CNS-0916972, CNS-0855128, and a gift from NEC.

9. References

[1] Amazon Auto Scaling - http://aws.amazon.com/autoscaling/

[2] Amazon RDS - http://aws.amazon.com/rds/

[3] C. Amza, E. Cecchet, Anupam Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel – *Specification and implementation of dynamic Web site benchmarks* – WWC, 2002.

[4] M. N. Bennani and D. A. Menasce – *Resource allocation for autonomic data centers using analytic performance models* – ICAC '05, Washington, DC, USA, 2005.

[5] J. Blancet – *Snapshots in Xen* – Online FAQ, https://zagnut.storeitoffsite.com/home/jim.blancet/FAQ/Snapshots%20in%20xen

[6] E. Cecchet, R. Singh, U. Sharma and P. Shenoy – *Dolly: Virtualization-driven Database Provisioning for the Cloud* – UMass Technical Report UM-CS-2010-006.

[7] E. Cecchet, G. Candea and A. Ailamaki – *Middleware-based Database Replication: The Gaps between Theory and Practice*. – ACM SIGMOD, June 10-12, 2008

[8] J. Chen, G.Soundararajan, C.Amza – *Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers* – ICAC '06, June 2006.

[9] S. Elnikety, S. Dropsho, E. Cecchet and W. Zwaenepoel – *Predicting Replicated Database Scalability from Standalone Database Profiling* – EuroSys, April 2009.

[10] S. Ghanbari, G. Soundararajan, J. Chen, and C. Amza – *Adaptive Learning of Metric Correlations for Temperature-Aware Database Provisioning* – ICAC, June 2007.

[11] J. Hellerstein, F. Zhang, and P. Shahabuddin – *An Approach to Predictive Detection for Service Management* – Proceedings of the 12th Conference on Systems and Network Management, 1999.

[12] K. Manassiev and C. Amza – *Scaling and Continuous Availability in Database Server Clusters through Multiversion Replication* – DSN 2007, June 2007.

[13] Oracle – Oracle Real Application Clusters 11g – Oracle Technical White Paper, April 2007.

[14] OpenNebula project. http://opennebula.org/

[15] M. Otey – *SQL Server vs. SQL Azure: Where SQL Azure is Limited* - SQL Server Magazine, August 2010.

[16] Sequoia Project. http://sourceforge.net/projects/sequoiadb/

[17] G. Soundararajan and C. Amza – *Online data migration for autonomic provisioning of databases in dynamic content web servers* – 2005 Conference of the Centre For Advanced Studies on Collaborative Research, Toronto, October 2005.

[18] TPC-W Benchmark, ObjectWeb implementation, http://jmob.objectweb.org/tpcw.html.

[19] Transaction Processing Council. http://www.tpc.org/.

[20] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal – *Dynamic Provisioning for Multi-tier Internet Applications* – ICAC-05, Seattle, June 2005.

[21] D. Villela, P. Pradhan, and D. Rubenstein – *Provisioning Servers in the Application Tier for E-commerce Systems* – IWQOS 2004, June 2004.

[22] Q. Zhang, L. Cherkasova, and E. Smirni – *A regression based analytic model for dynamic resource provisioning of multi-tier applications* – ICAC '07, Washington, DC, 2007.